

5 **ON THE FLY GENERATION OF MULTIMEDIA CODE FOR**
IMAGE PROCESSING

Background of the Invention

10

Field of the Invention

15

The invention relates to the processing of multimedia data with processors that feature multimedia instruction enhanced instruction sets. More particularly, the invention relates to a method and apparatus for generating processor instruction sequences for image processing routines that use multimedia enhanced instructions.

Description of the Prior Art

In general, most programs that use image processing routines with multimedia instructions do not use a general-purpose compiler for these parts of the program. These programs typically use assembly routines to process such data. A resulting problem is that the assembly routines must be added to the code manually. This step requires high technical skill, is time demanding, and is prone to introduce errors into the code.

25

5 In addition, different type of processors, (for example, Intel's Pentium I w/MMX
and Pentium II, Pentium III, Willamette, AMD's K-6 and AMD's K-7 aka. Athlon)
each use different multimedia command sets. Examples of different multimedia
command sets are MMX, SSE and 3DNow. Applications that use these
multimedia command sets must have separate assembly routines that are
10 specifically written for each processor type.

At runtime, the applications select the proper assembly routines based on the
processor detected. To reduce the workload and increase the robustness of the
code, these assembly routines are sometimes generated by a routine specific
15 source code generator during program development.

One problem with this type of programming is that the applications must have
redundant assembly routines which can process the same multimedia data, but
which are written for the different types of processors. However, only one

20 assembly routine is actually used at runtime. Because there are many
generations of processors in existence, the size of applications that use
multimedia instructions must grow to be compatible with all of these processors.

In addition, as new processors are developed, all new routines must be coded for
these applications so that they are compatible with the new processors. An
25 application that is released prior to the release of a processor is incompatible

5 with the processor unless it is first patched/rebuilt with the new assembly
routines.

It would be desirable to provide programs that use multimedia instructions which
are smaller in size. It would be desirable to provide an approach that adapts such
10 programs to future processors more easily

Summary of the Invention

In accordance with the invention, a method and apparatus for generating
assembly routines for multimedia instruction enhanced data is shown and
15 described.

An example of multimedia data that can be processed by multimedia instructions
are the pixel blocks used in image processing. Most image processing routines
operate on rectangular blocks of evenly sized data pieces (e.g. 16x16 pixel
20 blocks of 8 bit video during MPEG motion compensation). The image processing
code is described as a set of source blocks, destination blocks and data
manipulations. Each block has a start address, a pitch (distance in bytes
between two consecutive lines) and a data format. The full processing code
includes width and height as additional parameters. All of these parameters can
25 either be integer constants or arguments to the generated routine. All data
operations are described on SIMD data types. A SIMD data type is a basic data

5 type (e.g. signed byte, signed word, or unsigned byte) and a number or repeats
(e.g. 16 pixels for MPEG Macroblocks). The size of a block (source or
destination) is always the size of its SIMD data type times its width in horizontal
direction and the height in vertical direction.

10 In the presently preferred embodiment of the invention, an abstract image
generator inside the application program produces an abstract routine
representation of the code that operates on the multimedia data using SIMD
operations. A directed acyclic graph is a typical example of a generic version. A
translator then generates processor specific assembly code from the abstract
15 representation.

Brief Description of the Drawings

FIG. 1 is a block diagram of a computer system that may be used to implement a
20 method and apparatus embodying the invention for translating a multimedia
routine from its abstract representation generated by an abstract routine
generator inside the application's startup code into executable code using the
code generator.

Description of the Preferred Embodiment

In Fig.1 the startup code 11 of the application program 13, further referred to as the abstract routine generator, generates an abstract representation 15 of the multimedia routine represented by a data flow graph. This graph is then

10 translated by the code generator 17 into a machine specific sequence of instructions 19, typically including several SIMD multimedia instructions. The types of operations that can be present inside the data flow graph include add, sub, multiply, average, maximum, minimum, compare, and, or, xor, pack, unpack

15 and merge operations. This list is not exhaustive as there are operations currently performed by MMX, SSE and 3DNow for example, which are not listed.

If a specific command set does not support one of these operations, the CPU specific part of the code generator replaces it by a sequence of simpler instructions (e.g. the maximum instruction can be replaced by a pair of subtract and add instruction using saturation arithmetic).

The abstract routine generator generates an abstract representation of the code, commonly in the form of a directed acyclic graph during runtime. This allows the creation of multiple similar routines using a loop inside the image processing code 21 for linear arrays, or to generate routines on the fly depending on user interaction. *E.g.* the bi-directional MPEG 2 motion compensation can be implemented using a set of sixty-four different but very similar routines, that can be generated by a loop in the abstract image generator. Or an interactive paint

5 program can generate filters or pens in the form of abstract representations based on user input, and can use the routine generator to create efficient code sequences to perform the filtering or drawing operation. Examples of the data types processed by the code sequences include: SIMD input data, image input data and audio input data.

10

Examples of information provided by the graphs include the source blocks, the target blocks, the change in the block, color, stride, change in stride, display block, and spatial filtering.

15 The accuracy of the operation inside the graphs can be tailored to meet the requirements of the program. The abstract routine generator can increase its precision by increasing the level of arithmetics per pixel. For example, 7-bit processing can be stepped up to 8-bit, or 8-bit to 16-bit. *E.g.* motion compensation routines with different types of rounding precision can be
20 generated by the abstract routine generator.

The abstract representation, in this case the graph 15, is then sent to the translator 17 where it is translated into optimized assembly code 19. The translator uses standard compiler techniques to translate the generic graph structure into a specific sequence of assembly instructions. As the description is very generic, there is no link to a specific processor architecture, and because it
25

5 is very simple it can be processed without requiring complex compiler
techniques. This enables the translation to be executed during program startup
without causing a significant delay. Also, the abstract generator and the
translator do not have to be programmed in assembly. The CPU specific
translator may reside in a dynamic link library and can therefore be replaced if
10 the system processor is changed. This enables programs to use the multimedia
instructions of a new processor, without the need to be changed.

Tables A-C provide sample code that generates an abstract representation for a
motion compensation code that can be translated to an executable code
15 sequence using the invention.

TABLE A

```
20 #ifndef MPEG2MOTIONCOMPENSATION_H
#define MPEG2MOTIONCOMPENSATION_H

25 #include "driver\softwarecinemaster\common\prelude.h"
#include "..\..\BlockVideoProcessor\BVPXMMXCodeConverter.h"

30 // Basic block motion compensation functions
//
class MPEG2MotionCompensation
{
35 protected:
//
// Function prototype for a unidirectional motion compensation
routine
//
40     typedef void (_stdcall * CompensationCodeType) (BYTE * sourceBase,
int sourceStride,
                                BYTE * targetBase, short * deltaBase,
int deltaStride,
                                int num);
```

```

5
    //
    // Function prototype for a bidirectional motion compensation
    routine
    //
10   typedef void (_stdcall * BiCompensationCodeType)(BYTE *
    source1Base, BYTE * source2Base, int sourceStride,
                           BYTE * targetBase, short * deltaBase,
    int deltaStride,
                           int num);

15

    //
    // Motion compensation routines for unidirectional prediction.
    Each routine
20   // handles one case. The indices are
    // - y-uv : if it is luma data the index is 0 otherwise 1
    // - delta : error correction data is present (eg. the block
    is not skipped)
    // - halfy : half pel prediction is to be performed in
25   vertical direction
    // - halfx : half pel prediction is to be performed in
    horizontal direction
    //
30   CompensationCodeType      compensation[2][2][2][2];           //
    y-uv delta halfy halfx
    BVPCodeBlock      * compensationBlock[2][2][2][2];

    //
    // Motion compensation routines for bidirectional prediction.
35   Each routine
        // handles one case. The indices contain the same parameters as
        in the
        // unidirectional case, plus the half pel selectors for the
        second source
40   //
    BiCompensationCodeType    bicompensation[2][2][2][2][2][2];    //
    y-uv delta halfly half1x half2y half2x
    BVPCodeBlock      * bicompensationBlock[2][2][2][2][2][2];
    public:
45   //
    // Perform a unidirectional compensation.
    //
    void MotionCompensation(BYTE * sourcep, int stride, BYTE * destp,
    short * deltap, int dstride, int num, bool uv, bool delta, int halfx,
50   int halfy)
    {
        compensation[uv][delta][halfy][halfx](sourcep, stride, destp,
    deltap, dstride, num);
    }

55
    //
    // Perform bidirectional compensation
    //
    void BiMotionCompensation(BYTE * source1p, BYTE * source2p, int
60   stride, BYTE * destp, short * deltap, int dstride, int num, bool uv,
    bool delta, int half1x, int halfly, int half2x, int half2y)

```

```
5      {
6          bicompensation[uv][delta][half1y][half1x][half2y][half2x](source1p,
7          source2p, stride, destp, deltap, dstride, num);
8      }
9
10     MPEG2MotionCompensation(void);
11     ~MPEG2MotionCompensation(void);
12 };
13
14 #endif
```

TABLE B

```

20 #include "MPEG2MotionCompensation.h"
25 #include "...\\BlockVideoProcessor\\BVPXMMXCodeConverter.h"
30
35 // Create the dataflow to fetch a data element from a source block,
40 // with or without half pel compensation in horizontal and/or
45 // vertical direction.
50 // BVPDataSourceInstruction * BuildBlockMerge(BVPSourceBlock *
55 source1BlockA,
60 source1BlockB,
65 source1BlockC,
70 source1BlockD,
75 int halfx, int halfy)
80 {
85     if (halfy)
90     {
95         if (halfx)
100     {
105         // Half pel prediction in h and v direction, the graph part
110 looks like this
115
120     //           .-- (LOAD source1BlockA)
125     //           /
130     //           .-- (AVG)
135     //           /   \
140     //           /     \-- (LOAD source1BlockB)
145     //           \-- (AVG)   \
150     //           \   /     .-- (LOAD source1BlockC)
155     //           \ /     \
160     //           \     \-- (AVG)   \
165     //           \   /     \
170     //           \ /     \-- (LOAD source1BlockD)
175     //
180     return new BVPDataOperation
185 }

```



```

5           // Full pel prediction
6           //
7           // <--(LOAD source1BlockA)
8           //
9           return new BVPDataLoad(source1BlockA);
10          }
11      }
12  }

MPEG2MotionCompensation::MPEG2MotionCompensation(void)
15  {
16      int yuv, delta, halfy, halfx, halfly, halflx, half2y, half2x;
17      BVPBlockProcessor * bvp;
18      BVPCodeBlock * code;

19      BVPArgument * source1Base;
20      BVPArgument * source2Base;
21      BVPArgument * sourceStride;
22      BVPArgument * targetBase;
23      BVPArgument * deltaBase;
24      BVPArgument * deltaStride;
25      BVPArgument * height;

26      BVPSourceBlock * source1BlockA;
27      BVPSourceBlock * source1BlockB;
28      BVPSourceBlock * source1BlockC;
29      BVPSourceBlock * source1BlockD;
30      BVPSourceBlock * source2BlockA;
31      BVPSourceBlock * source2BlockB;
32      BVPSourceBlock * source2BlockC;
33      BVPSourceBlock * source2BlockD;

34      BVPSourceBlock * deltaBlock;
35      BVPTargetBlock * targetBlock;

36      BVPDataSourceInstruction * postMC;
37      BVPDataSourceInstruction * postCorrect;
38      BVPDataSourceInstruction * deltaData;

39      //
40      // Build unidirectional motion compensation routines
41      //
42      for(yuv = 0; yuv<2; yuv++)
43      {
44          for(delta=0; delta<2; delta++)
45          {
46              for(halfy=0; halfy<2; halfy++)
47              {
48                  for(halfx=0; halfx<2; halfx++)
49                  {
50                      bvp = new BVPBlockProcessor();

51                      bvp->AddArgument(height           = new BVPArgument(false));
52                      bvp->AddArgument(deltaStride      = new BVPArgument(false));
53                      bvp->AddArgument(deltaBase        = new BVPArgument(true));
54                      bvp->AddArgument(targetBase       = new BVPArgument(true));
55                      bvp->AddArgument(sourceStride     = new BVPArgument(false));

```

```

5           bvp->AddArgument(source1Base      = new BVPArgument(true));

        //
        // Width is always sixteen pixels, so one vector of sixteen
10      unsigned eight bit elements,
        // height may vary, therefore it is an argument
        //
        bvp->SetDimension(1, height);

        //
15      // Four potential source blocks, B is one pel to the right,
      C one down and D right and down
        //
        bvp->AddSourceBlock(source1BlockA = new
BVPSourceBlock(source1Base,
20      sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
        bvp->AddSourceBlock(source1BlockB = new
BVPSourceBlock(BVPPointer(source1Base, 1 + yuv),
      sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
        bvp->AddSourceBlock(source1BlockC = new
25      BVPSourceBlock(BVPPointer(source1Base, sourceStride, 1, 0),
      sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
        bvp->AddSourceBlock(source1BlockD = new
BVPSourceBlock(BVPPointer(source1Base, sourceStride, 1, 1 + yuv),
      sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));

30      //
        // If we have error correction data, we need this source
      block as well
        //
35      if (delta)
        bvp->AddSourceBlock(deltaBlock  = new
BVPSourceBlock(deltaBase, deltaStride, BVPDataFormat(BVPDT_S16, 16),
      0x10000));

40      //
        // The target block to write the data into
        //
        bvp->AddTargetBlock(targetBlock = new
BVPTargetBlock(targetBase, sourceStride, BVPDataFormat(BVPDT_U8, 16),
45      0x10000));

        //
        // Load a source block base on the half pel settings
        //
50      bvp->AddInstruction(postMC = BuildBlockMerge(source1BlockA,
      source1BlockB, source1BlockC, source1BlockD, halfx, halfy));

        if (delta)
        {
55      deltaData = new BVPDataLoad(deltaBlock);

        if (yuv)
        {
        //
60      // It is chroma data and we have error correction data.
      The u and v

```

```

5           // parts have to be interleaved, therefore we need the
merge instruction
10          //
11          //
12          //           .-- (CONV S16) <--postMC
13          //           /           \
14          // <-- (CONV U8) <-- (ADD)           \
15          //           \           .-- (SPLIT H) <-- \
16          //           \           /           \
17          //           -- (MERGE OE)           \
18          >-- (LOAD delta)
19          //
20          //
21          bvp->AddInstruction
22          (
23          postCorrect =
24          new BVPDataConvert
25          (
26          BVPDT_U8,
27          new BVPDataOperation
28          (
29          BVPDO_ADD,
30          new BVPDataConvert
31          (
32          BVPDT_S16,
33          postMC
34          ),
35          new BVPDataMerge
36          (
37          BVPDM_ODDEVEN,
38          new BVPDataSplit
39          (
40          BVPDS_HEAD,
41          deltaData
42          ),
43          new BVPDataSplit
44          (
45          BVPDS_TAIL,
46          deltaData
47          )
48          )
49          )
50          );
51      }
52      else
53      {
54      //
55      // It is luma data with error correction
56      //
57      //           .-- (CONV S16) <--postMC
58      //           /           \
59      // <-- (CONV U8) <-- (ADD)           \
60      //           \           .-- (LOAD delta)
61      //
62      bvp->AddInstruction

```

```

5
6         (
7             postCorrect =
8                 new BVPDataConvert
9                     (
10                         BVPDT_U8,
11                         new BVPDataOperation
12                             (
13                                 BVPDO_ADD,
14                                 new BVPDataConvert
15                                     (
16                                         BVPDT_S16,
17                                         postMC
18                                     ),
19                                     deltaData
20                                 )
21             );
22         }
23
24
25         // Store into the target block
26         // (STORE targetBlock)<--...
27         // bvp->AddInstruction
28             (
29                 new BVPDataStore
30                     (
31                         targetBlock,
32                         postCorrect
33                     )
34             );
35         }
36     else
37     {
38         // No error correction data, so store motion result into
39         target block
40         // (STORE targetBlock)<--...
41         // bvp->AddInstruction
42             (
43                 new BVPDataStore
44                     (
45                         targetBlock,
46                         postMC
47                     )
48             );
49         }
50
51         BVPXMMXCodeConverter conv;
52
53         //
54         // Convert graph into machine language
55         //

```

```

5           compensationBlock[yuv][delta][halfy][halfx] = code =
conv.Convert(bvp);

10          //
11          // Get function entry pointer
12          //
13          compensation[yuv][delta][halfy][halfx] =
(CompensationCodeType)(code->GetCodeAddress());

15          //
16          // delete graph
17          //
18          delete bvp;
19          }

20      }

25      //
26      // build motion compensation routines for bidirectional prediction
27      //
28      for(yuv = 0; yuv<2; yuv++)
29      {
30          for(delta=0; delta<2; delta++)
31          {
32              for(halfly=0; halfly<2; halfly++)
33              {
34                  for(halflx=0; halflx<2; halflx++)
35                  {
36                      for(half2y=0; half2y<2; half2y++)
37                      {
38                          for(half2x=0; half2x<2; half2x++)
39                          {
40                              bvp = new BVPBlockProcessor();
41
42                                  bvp->AddArgument(height) = new
43 BVPArgument(false));
44                                  bvp->AddArgument(deltaStride) = new
45 BVPArgument(false));
46                                  bvp->AddArgument(deltaBase) = new
47 BVPArgument(true));
48                                  bvp->AddArgument(targetBase) = new
49 BVPArgument(true));
50                                  bvp->AddArgument(sourceStride) = new
51 BVPArgument(false));
52                                  bvp->AddArgument(source2Base) = new
53 BVPArgument(true));
54                                  bvp->AddArgument(source1Base) = new
55 BVPArgument(true));
56
57                                  bvp->SetDimension(1, height);
58
59                                  //
60                                  // We now have two source blocks, so we need eight
blocks for the half pel
61                                  // prediction
62                                  //

```

```

5           bvp->AddSourceBlock(source1BlockA = new
  BVPSourceBlock(source1Base,
  sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
           bvp->AddSourceBlock(source1BlockB = new
  BVPSourceBlock(BVPPointer(source1Base, 1 + yuv),
10      sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
           bvp->AddSourceBlock(source1BlockC = new
  BVPSourceBlock(BVPPointer(source1Base, sourceStride, 1, 0),
  sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
           bvp->AddSourceBlock(source1BlockD = new
15      BVPSourceBlock(BVPPointer(source1Base, sourceStride, 1, 1 + yuv),
  sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
           bvp->AddSourceBlock(source2BlockA = new
  BVPSourceBlock(source2Base,
  sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
20           bvp->AddSourceBlock(source2BlockB = new
  BVPSourceBlock(BVPPointer(source2Base, 1 + yuv),
  sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
           bvp->AddSourceBlock(source2BlockC = new
  BVPSourceBlock(BVPPointer(source2Base, sourceStride, 1, 0),
25      sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));
           bvp->AddSourceBlock(source2BlockD = new
  BVPSourceBlock(BVPPointer(source2Base, sourceStride, 1, 1 + yuv),
  sourceStride, BVPDataFormat(BVPDT_U8, 16), 0x10000));

30           if (delta)
           bvp->AddSourceBlock(deltaBlock = new
  BVPSourceBlock(deltaBase, deltaStride, BVPDataFormat(BVPDT_S16, 16),
  0x10000));

35           bvp->AddTargetBlock(targetBlock = new
  BVPTargetBlock(targetBase, sourceStride, BVPDataFormat(BVPDT_U8, 16),
  0x10000));

40           //
           // Build bidirectional prediction from two
           unidirectional predictions
           //
           //           .--BuildBlockMerge(source1Block*)
           //           /
45           // <-- (AVG)
           //           \
           //           '--BuildBlockMerge(source2Block*)
           //
           bvp->AddInstruction
50           (
           postMC =
           new BVPDataOperation
           (
             BVPDO_AVG,
             BuildBlockMerge(source1BlockA, source1BlockB,
55             source1BlockC, source1BlockD, half1x, half1y),
             BuildBlockMerge(source2BlockA, source2BlockB,
             source2BlockC, source2BlockD, half2x, half2y)
           )
           );

```

```

5
// Apply error correction, see unidirectional case
//
10    if (delta)
    {
        deltaData = new BVPDataLoad(deltaBlock);

        if (yuv)
        {
            bvp->AddInstruction
            (
            postCorrect =
            new BVPDataConvert
            (
            BVPDT_U8,
            new BVPDataOperation
            (
            BVPDO_ADD,
            new BVPDataConvert
            (
            BVPDT_S16,
            postMC
            ),
            new BVPDataMerge
            (
            BVPDM_ODDEVEN,
            new BVPDataSplit
            (
            BVPDS_HEAD,
            deltaData
            ),
            new BVPDataSplit
            (
            BVPDS_TAIL,
            deltaData
            )
            )
            )
        );
    }
    else
    {
        bvp->AddInstruction
        (
        postCorrect =
        new BVPDataConvert
        (
        BVPDT_U8,
        new BVPDataOperation
        (
        BVPDO_ADD,
        new BVPDataConvert
        (
        BVPDT_S16,
        postMC
        ),
        )
    );
}
}

```

```

5                         deltaData
                           )
                           )
                           );
                           }
10
                           bvp->AddInstruction
                           (
                           new BVPDataStore
                           (
                           targetBlock,
                           postCorrect
                           )
                           );
                           }
20
                           else
                           {
                           bvp->AddInstruction
                           (
                           new BVPDataStore
                           (
                           targetBlock,
                           postMC
                           )
                           );
                           }
25
                           }
30
                           BVPXMMXCodeConverter conv;

                           //
                           // Translate routines
                           //

                           bicompenstationBlock[yuv][delta][halfly][halflx][half2y][half2x] =
                           code = conv.Convert(bvp);
40

                           bicompenstation[yuv][delta][halfly][halflx][half2y][half2x] =
                           (BiCompensationCodeType) (code->GetCodeAddress());

                           delete bvp;
                           }
                           }
                           }
                           }
                           }
                           }
                           }
                           }

                           MPEG2MotionCompensation::~MPEG2MotionCompensation(void)
55
                           {
                           int yuv, delta, halfy, halfx, halfly, halflx, half2y, half2x;

                           //
                           // free all motion compensation routines
                           //
60
                           for(yuv = 0; yuv<2; yuv++)

```

```

5      {
10     for(delta=0; delta<2; delta++)
15     {
20       for(halfy=0; halfy<2; halfy++)
25       {
30         for(halfx=0; halfx<2; halfx++)
35         {
40           delete compensationBlock[yuv][delta][halfy][halfx];
        }
      }
    }
  }
}
for(yuv = 0; yuv<2; yuv++)
{
  for(delta=0; delta<2; delta++)
  {
    for(halfly=0; halfly<2; halfly++)
    {
      for(halflx=0; halflx<2; halflx++)
      {
        for(half2y=0; half2y<2; half2y++)
        {
          for(half2x=0; half2x<2; half2x++)
          {
            delete
30   bicompenstationBlock[yuv][delta][halfly][halflx][half2y][half2x];
          }
        }
      }
    }
  }
}
}

```

TABLE C

```

40
45 #ifndef BVPGENERIC_H
#define BVPGENERIC_H

50 #include "BVPList.h"

  //
  // Argument descriptor.  An argument can be either a pointer or an
  integer used
55 // as a stride, offset or width/height value.
  //
  class BVPArgument
  {
    public:
      bool pointer;
      int index;
  }

```

```

5      BVPArgument(bool pointer_)
        : pointer(pointer_), index(0) {}
    };

10     //
11     // Description of an integer value used as a stride or offset. An
12     integer value
13     // can be either an argument or a constant
14     //
15     class BVPInteger
16     {
17     public:
18         int           value;
19         BVPArgument * arg;
20
21         BVPInteger(void)
22             : value(0), arg(NULL) {}
23         BVPInteger(int value_)
24             : value(value_), arg(NULL) {}
25         BVPInteger(unsigned value_)
26             : value((int)value_), arg(NULL) {}
27         BVPInteger(BVPArgument * arg_)
28             : value(0), arg(arg_) {}

29         bool operator==(BVPInteger i2)
30             {
31                 return arg ? (i2.arg == arg) : (i2.value == value);
32             }
33     };
34
35     //
36     // Description of a memory pointer used as a base for source and
37     target blocks.
38     // A pointer can be a combination of an pointer base, a constant
39     offset and
40     // a variable index with scaling
41     //
42     class BVPPointer
43     {
44     public:
45         BVPArgument * base;
46         BVPArgument * index;
47         int           offset;
48         int           scale;
49
50         BVPPointer(BVPArgument * base_)
51             : base(base_), index(NULL), offset(0), scale(0) {}

52         BVPPointer(BVPPointer base_, int offset_)
53             : base(base_.base), index(NULL), offset(offset_), scale(0) {}

54         BVPPointer(BVPPointer base_, BVPInteger index_, int scale_, int
55         offset_)
56             : base(base_.base), index(index_.arg), offset(offset_), scale(scale_) {}
57     };

```

```

5
//
// Base data formats for scalar types
//
10 enum BVPBaseDataFormat
{
    BVPDT_U8,    // Unsigned 8 bits
    BVPDT_U16,   // Unsigned 16 bits
    BVPDT_U32,   // Unsigned 32 bits
    BVPDT_S8,    // Signed 8 bits
15    BVPDT_S16,   // Signed 16 bits
    BVPDT_S32    // Signed 32 bits
};

//
20 // Data format descriptor for scalar and vector (multimedia SIMD)
types
    // Each data type is a combination of a base type and a vector size.
    // Scalar types are represented by a vector size of one.
    //
25 class BVPDataFormat
{
public:
    BVPBaseDataFormat format;
    int num;
30
    BVPDataFormat(BVPBaseDataFormat _format, int _num = 1)
        : format(_format), num(_num) {}

    BVPDataFormat(void)
35        : format(BVPDT_U8), num(0) {}

    BVPDataFormat(BVPDataFormat & f)
        : format(f.format), num(f.num) {}

40    BVPDataFormat operator* (int times)
        {return BVPDataFormat(format, num * times);}

    BVPDataFormat operator/ (int times)
45        {return BVPDataFormat(format, num / times);}

    int BitsPerElement(void) {static const int sz[] = {8, 16, 32, 8,
16, 32}; return sz[format];}
    int BitsPerChunk(void) {return BitsPerElement() * num;}
};

50
//
// Operation codes for binary data operations that have the
// same operand type for both sources and the destination
//
55 enum BVPDataOperationCode
{
    BVPDO_ADD,           // add with wraparound
    BVPDO_ADD_SATURATED, // add with saturation
    BVPDO_SUB,           // subtract with wraparound
60    BVPDO_SUB_SATURATED, // subtract with saturation
    BVPDO_MAX,           // maximum
};

```

```

5      BVPDO_MIN,           // minimum
     BVPDO_AVG,           // average (includes rounding towards nearest)
     BVPDO_EQU,           // equal
     BVPDO_OR,            // binary or
     BVPDO_XOR,           // binary exclusive or
10    BVPDO_AND,           // binary and
     BVPDO_ANDNOT,        // binary and not
     BVPDO_MULL,          // multiply keep lower half
     BVPDO_MULH           // multiply keep upper half
    };

15
    //
    // Operations that extract a part of a data element
    //
20  enum BVPDataSplitCode
    {
        BVPDS_HEAD,          // extract first half
        BVPDS_TAIL,           // extract second half
        BVPDS_ODD,            // extract odd elements
        BVPDS_EVEN            // extract even elements
    };

25
    //
    // Operations that combine two data elements
    //
30  enum BVPDataMergeCode
    {
        BVPDM_UPPERLOWER,    // chain first and second operands
        BVPDM_ODDEVEN        // interleave first and second operands
    };

35
    //
    // Node types in the data flow graph
    //
40  enum BVPInstructionType
    {
        BVPIT_LOAD,           // load an element from a source block
        BVPIT_STORE,           // store an element into a source block
        BVPIT_CONSTANT,        // load a constant value
        BVPIT_SPLIT,           // split an element
45    BVPIT_MERGE,           // merge two elements
        BVPIT_CONVERT,         // perform a data conversion
        BVPIT_OPERATION         // simple binary data operation
    };

50
    //
    // Descriptor of a data block. Contains a base pointer, a
    stride(pitch), a
    // format and an incrementor in vertical direction. The vertical
    block position
55    // can be incremented by a fraction or a multiple of the given pitch.
    //
    class BVPBlock
    {
        public:
60        BVPPointer    base;
        BVPInteger    pitch;

```

```

5      BVPDataFormat format;
6      int         yscale;
7      int         index;

8      BVPBlock(BVPPointer _base, BVPInteger _pitch, BVPDataFormat
9      _format, int _yscale)
10     : base(_base), pitch(_pitch), format(_format), yscale(_yscale)
11     {}
12     };

13
14     //
15     // Descriptor of a source block
16     //
17     class BVPSourceBlock : public BVPBlock
18     {
19     public:
20         BVPSourceBlock(BVPPointer base, BVPInteger pitch, BVPDataFormat
21         format, int yscale)
22             : BVPBlock(base, pitch, format, yscale) {}
23         };
24
25     //
26     // Descriptor of a target block
27     //
28     class BVPTargetBlock : public BVPBlock
29     {
30     public:
31         BVPTargetBlock(BVPPointer base, BVPInteger pitch, BVPDataFormat
32         format, int yscale)
33             : BVPBlock(base, pitch, format, yscale) {}
34         };
35

36     class BVPDataSource;
37     class BVPDataDrain;
38     class BVPDataInstruction;
39
40     //
41     // Source connection element of a node in the data flow graph.  Each
42     node in
43     // the graph contains one or none source connection.  A source
44     connection is
45     // the output of a node in the graph.  Each source connection can be
46     connected
47     // to any number of drain connections in other nodes of the flow
48     graph.  The
49     // source is the output side of a node.
50     //
51     class BVPDataSource
52     {
53     public:
54         BVPDataFormat      format;
55         BVPList<BVPDataDrain *>    drain;

56         BVPDataSource(BVPDataFormat _format) : format(_format) {}

57         virtual void AddInstructions(BVPList<BVPDataInstruction *> &
58         instructions) {}

```

```
5     virtual BVPDataInstruction * ToInstruction(void) {return NULL;}
6     };
7
8     // Drain connection element of a node in the data flow graph. Each
9     node
10    // can have none, one or two drain connections (but only one drain
11    object
12    // to represent both). Each drain connects to exactly one source on
13    the
14    // target side. As eachnode can have only two inputs, each drain is
15    connected
16    // (through the node) with two sources. The drain is the input side
17    of a
18    // node.
19    //
20    class BVPDataDrain
21    {
22    public:
23        BVPDataSource      * source1;
24        BVPDataSource      * source2;
25
26        BVPDataDrain(BVPDataSource * source1_, BVPDataSource * source2_ =
27        NULL)
28            : source1(source1_), source2(source2_) {}
29
30        virtual BVPDataInstruction * ToInstruction(void) {return NULL;}
31    };
32
33    //
34    // Each node in the graph represents one abstract instruction. It
35    has an
36    // instruction type that describes the operation of the node.
37    //
38    class BVPDataInstruction
39    {
40    public:
41        BVPInstructionType type;
42        int                 index;
43
44        BVPDataInstruction(BVPInstructionType type_)
45            : type(type_), index(-1) {}
46
47        virtual ~BVPDataInstruction(void) {}
48
49        virtual void AddInstructions(BVPList<BVPDataInstruction *> &
50        instructions);
51        virtual void GetOperationBits(int & minBits, int & maxBits);
52
53        virtual BVPDataFormat GetInputFormat(void) = 0;
54        virtual BVPDataFormat GetOutputFormat(void) = 0;
55
56        virtual BVPDataSource * ToSource(void) {return NULL;}
57        virtual BVPDataDrain * ToDrain(void) {return NULL;}
58    };
59
60    //
```

```
5     // Node that is a data source
  // 
10    class BVPDataSourceInstruction : public BVPDataInstruction, public
BVPDataSource
  {
15    public:
      BVPDataSourceInstruction(BVPInstructionType type_, BVPDataFormat
format_)
        : BVPDataInstruction(type_), BVPDataSource(format_) {}

20    void GetOperationBits(int & minBits, int & maxBits);

      BVPDataFormat GetOutputFormat(void) {return format;}
      BVPDataFormat GetInputFormat(void) {return format;}

25    BVPDataInstruction * ToInstruction(void) {return this;}
      BVPDataSource * ToSource(void) {return this;}
  };

  //
25  // Node that is a data source and has one or two sources connected to
its drain
  //
30  class BVPDataSourceDrainInstruction : public BVPDataSourceInstruction,
public BVPDataDrain
  {
35  public:
      BVPDataSourceDrainInstruction(BVPInstructionType type_,
BVPDataFormat format_, BVPDataSource * source1_)
        : BVPDataSourceInstruction(type_, format_),
BVPDataDrain(source1_)
        {source1->drain.Insert(this);}

      BVPDataSourceDrainInstruction(BVPInstructionType type_,
BVPDataFormat format_, BVPDataSource * source1_, BVPDataSource *
source2_)
        : BVPDataSourceInstruction(type_, format_),
BVPDataDrain(source1_, source2_)
        {source1->drain.Insert(this);source2->drain.Insert(this);}
  };

45  //
  // Instruction to load data from a source block
  //
50  class BVPDataLoad : public BVPDataSourceInstruction
  {
55  public:
      BVPSourceBlock * block;
      int offset;

      BVPDataLoad(BVPSourceBlock * block_, int offset_ = 0)
        : BVPDataSourceInstruction(BVPIT_LOAD, block_->format),
block(block_), offset(offset_) {}

60  void AddInstructions(BVPList<BVPDataInstruction *> & instructions);
  };
  //
```

```

5      // Instruction to store data into a target block
6      //
7      class BVPDataStore : public BVPDataInstruction, public BVPDataDrain
8      {
9      public:
10         BVPTargetBlock * block;
11
12         BVPDataStore(BVPTargetBlock * block_, BVPDataSource * source)
13             : BVPDataInstruction(BVPIT_STORE), BVPDataDrain(source),
14               block(block_)
15             {source->drain.Insert(this);}
16
17         void AddInstructions(BVPList<BVPDataInstruction *> & instructions);
18
19         BVPDataFormat GetOutputFormat(void) {return source1->format;}
20         BVPDataFormat GetInputFormat(void) {return source1->format;}
21
22         BVPDataInstruction * ToInstruction(void) {return this;}
23         BVPDataDrain * ToDrain(void) {return this;}
24     };
25
26     //
27     // Instruction to load a constant
28     //
29     class BVPDataConstant : public BVPDataSourceInstruction
30     {
31     public:
32         int value;
33
34         BVPDataConstant(BVPDataFormat format, int value_)
35             : BVPDataSourceInstruction(BVPIT_CONSTANT, format),
36               value(value_) {}
37
38         //
39         // Instruction to split a data element
40         //
41         class BVPDataSplit : public BVPDataSourceDrainInstruction
42         {
43         public:
44             BVPDataSplitCode code;
45
46             BVPDataSplit(BVPDataSplitCode code_, BVPDataSource * source)
47                 : BVPDataSourceDrainInstruction(BVPIT_SPLIT, source->format / 2,
48                   source), code(code_) {}
49
50             void AddInstructions(BVPList<BVPDataInstruction *> & instructions);
51
52             BVPDataDrain * ToDrain(void) {return this;}
53
54             BVPDataFormat GetInputFormat(void) {return source1->format;}
55         };
56
57         //
58         // Instruction to merge two data elements
59         //
60         class BVPDataMerge : public BVPDataSourceDrainInstruction

```

```

5      {
6      public:
7          BVPDataMergeCode code;
8
9          BVPDataMerge(BVPDataMergeCode code_, BVPDataSource * source1_,
10         BVPDataSource * source2_)
11             : BVPDataSourceDrainInstruction(BVPIT_MERGE, source1_->format *
12             2, source1_, source2_), code(code_) {}
13
14
15         void AddInstructions(BVPList<BVPDataInstruction *> & instructions);
16
17             BVPDataDrain * ToDrain(void) {return this;}
18
19             BVPDataFormat GetInputFormat(void) {return source1->format;}
20         };
21
22         //
23         // Instruction to convert the basic vector elements of an data
24         element into
25         // a different format (eg. from signed 16 bit to unsigned 8 bits).
26         //
27         class BVPDataConvert : public BVPDataSourceDrainInstruction
28         {
29             public:
30                 BVPDataConvert(BVPBaseDataFormat target, BVPDataSource * source)
31                     : BVPDataSourceDrainInstruction(BVPIT_CONVERT,
32                     BVPDataFormat(target, source->format.num), source) {}
33
34
35                 void AddInstructions(BVPList<BVPDataInstruction *> & instructions);
36
37                 BVPDataDrain * ToDrain(void) {return this;}
38
39                 BVPDataFormat GetInputFormat(void) {return source1->format;}
40             };
41
42         //
43         // Basic data manipulation operation from two sources to one drain.
44         //
45         class BVPDataOperation : public BVPDataSourceDrainInstruction
46         {
47             public:
48                 BVPDataOperationCode code;
49
50                 BVPDataOperation(BVPDataOperationCode code_, BVPDataSource *
51                     source1_, BVPDataSource * source2_)
52                     : BVPDataSourceDrainInstruction(BVPIT_OPERATION, source1_-
53                     >format, source1_, source2_), code(code_) {}
54
55                 void AddInstructions(BVPList<BVPDataInstruction *> & instructions);
56
57                 BVPDataDrain * ToDrain(void) {return this;}
58             };
59
60         //
61         // Descriptor for one image block processing routine. It contains
62         the arguments, the

```

```

5      // size and the dataflow graph.  On destruction of the block
processor all argument,
     // blocks and instructions are also deleted.
     //
10     class BVPBlockProcessor
11     {
12     public:
13         BVPInteger  width;
14         BVPInteger  height;
15
16         BVPList<BVPBlock *> blocks;
17         BVPList<BVPDataInstruction *> instructions;
18         BVPList<BVPArgument *> args;
19
20         BVPBlockProcessor(void)
21         {
22         }
23
24         ~BVPBlockProcessor(void);
25
26         //
27         // Add an argument to the list of arguments.  Please note that
28         the arguments
29             // are added in the reverse order of the c-calling convention.
30             //
31         void AddArgument(BVPArgument * arg)
32         {
33             arg->index = args.Num();
34             args.Insert(arg);
35         }
36
37         //
38         // Set the dimension of the operation rectangle.  The width and
39         height can
40             // either be constants or arguments to the routine.
41             //
42         void SetDimension(BVPInteger width, BVPInteger height)
43         {
44             this->width = width;
45             this->height = height;
46         }
47
48         //
49         // Add a source block to the processing
50         //
51         void AddSourceBlock(BVPSourceBlock * block)
52         {
53             block->index = blocks.Num();
54             blocks.Insert(block);
55         }
56
57         //
58         // Add a target block to the processing
59         //
60         void AddTargetBlock(BVPTargetBlock * block)
61         {
62             block->index = blocks.Num();

```

```
5         blocks.Insert(block);
    }

    //
    // Add an instruction to the dataflow graph.  All referenced
10  instructions
        // will also be added to the graph if they are not yet part of
    it.
    //
    void AddInstruction(BVPDataInstruction * ins)
15  {
    ins->AddInstructions(instructions);
    }

    void GetOperationBits(int & minBits, int & maxBits);
20  }

#endif
```

25

Although the invention is described herein with reference to the preferred embodiment, one skilled in the art will readily appreciate that other applications may be substituted for those set forth herein without departing from the spirit and scope of the present invention. Accordingly, the invention should only be limited 30 by the claims included below.